A Coinductive Calculus for Asynchronous Side-effecting Processes

Sergey Goncharov and Lutz Schröder

Safe and Secure Cognitive Systems, DFKI GmbH, Bremen

Abstract. We present an abstract framework for concurrent processes in which atomic steps have generic side effects, handled according to the principle of monadic encapsulation of effects. Processes in this framework are potentially infinite resumptions, modelled using final coalgebras over the monadic base. As a calculus for such processes, we introduce a concurrent extension of Moggi's monadic metalanguage of effects. We establish soundness and completeness of a natural equational axiomatisation of this calculus. Moreover, we identify a corecursion scheme that is explicitly definable over the base language and provides flexible expressive means for the definition of new operators on processes, such as parallel composition. As a worked example, we prove the safety of a generic mutual exclusion scheme using a verification logic built on top of the equational calculus.

1 Introduction

Imperative programming languages work with many different side effects, such as I/O, state, exceptions, and others, which all moreover come with strong degrees of variations in detail. This variety is unified by the principle of monadic encapsulation of side-effects [21], which not only underlies extensive work in semantics (e.g. [17]) but, following [35], forms the basis for the treatment of side-effects in functional languages such as Haskell [25] and F# [31]. Monads do offer support for concurrent programming, in particular through variants of the *resumption monad transformer* [4, 13], which lifts resumptions in the style of Hennessy and Plotkin [15] to the monadic level, and which has moreover been used in information flow security [14], semantics of functional logic programming languages such as Curry [33], modelling underspecification of compilers, e.g. for ANSI C [23, 24], and to model the semantic of the π-calculus [10]. However, the formal basis for concurrent functional-imperative programming is not as well-developed as for the sequential case; in particular, Moggi's original *computational meta-language* is essentially limited to linear sequential monadic programs, and does not offer native support for concurrency.

The objective of the present work is to develop an extension of the computational meta-language that can serve as a minimal common basis for concurrent functional-imperative programming and semantics. We define an abstract *meta-calculus for monadic processes* that is based on the resumption monad transformer, and hence generic over the base effects inherent in individual process steps. We work with infinite resumptions, which brings tools from coalgebra into play, in particular corecursion

and coinduction [28]. We present a complete equational axiomatization of our calculus which includes a simple loop construct (in coalgebraic terms, coiteration) and then derive a powerful *corecursion schema* that allows defining processes by systems of equations. It has a fully syntactic justification, i.e. one can explicitly construct a solution to a corecursive equation system by means of the basic term language. Although there are strong corecursion results available in the literature (e.g. [1, 34, 19]), our corecursion schema does not seem to be covered by these, in particular as it permits prefixing corecursive calls with monadic sequential composition.

We exemplify our corecursion scheme with the definition of a number of basic imperative programming and process-algebraic primitives including parallel composition, and present a worked example, in which we outline a safety proof for a monadic version of Dekker's mutual exclusion algorithm, i.e. a concurrent algorithm with generic side-effects. To this end, we employ a more high-level verification logic that we develop on top of the basic equational calculus.

Further related work There is extensive work on axiomatic perspectives on effectful iteration and recursion, including traced pre-monoidal categories [2], complete iterative algebras [19], Kleene monads [11], and recursive monadic binding [8]. The abstract notion of resumption goes back at least to [15]. (Weakly) final coalgebras of I/O-trees have been considered in the context of dependent type theory for functional programming, without, however, following a fully parametrised monadic approach as pursued here [12]. A framework where infinite resumptions of a somewhat different type than considered here form the morphisms of a category of processes, which is thus enriched over coalgebras for a certain functor, is studied in [18], but no metalanguage is provided for such processes. A metalanguage that essentially adds least fixed points, i.e. *inductive* data types as opposed to *coinductive* process types as used in the present work, to Moggi's base language is studied in [9]; reasoning principles in this framework are necessarily of a rather different flavour. A resumption monad without a base effect, the *delay monad*, is studied in [3] with a view to capturing general recursion. Our variant of the resumption monad transformer belongs to the class of *parametrized monads* introduced in [34], where a form of corecursive scheme is established which however does not seem to cover the one introduced here.

2 Computational Monads and Resumptions

We briefly recall the basic concepts of the monadic representation of side-effects, and then present the specific semantic framework required for the present work. Intuitively, a monad $\mathbb{T}$ (mostly referred to just as T) associates to each type A a type TA of computations with results in A; a function with side effects that takes inputs of type A and returns values of type B is, then, just a function of type $A \to TB$. In other words, by means of a monad we can abstract from notions of computation by switching from non-pure functions to pure ones with a converted type profile. One of the equivalent ways to define a monad over a category $\mathbf{C}$ is by giving a *Kleisli triple* $\mathbb{T} = (T, \eta, \_^\star)$ where $T : \mathrm{Ob}\,\mathbf{C} \to \mathrm{Ob}\,\mathbf{C}$ is a function, η is a family of morphisms $\eta_A : A \to TA$ called *unit*, and $\_^\star$ assigns to each morphism $f : A \to TB$ a morphism $f^\star : TA \to TB$ such that

$\eta_A^\star = \mathrm{id}_{TA}$, $\quad f^\star \circ \eta_A = f$, $\quad$ and $\quad g^\star \circ f^\star = (g^\star \circ f)^\star$. Thus, η_A converts values of type A into side-effect free computations, and $\_^\star$ supports the sequential composition $g^\star f$ of programs $f : A \to TB$ and $g : B \to TC$. A monad over a Cartesian category is *strong* if it is equipped with a natural transformation $\tau_{A,B} : A \times TB \to T(A \times B)$ called *strength*, subject to certain coherence conditions [21]. Since we are interested in concurrency, we require additional structure for non-determinism:

Definition 1 (Strong semi-additive monads). A strong monad $\mathbb{T} = (T, \eta, \_^\star, \tau)$ is *semi-additive* if there exist natural transformations $\delta : 1 \to T$ and $\varpi : T \times T \to T$ making every TA an internal bounded join-semilattice object so that $\_^\star$ and τ respect the join-semilattice structure in the following sense:

$$
\begin{aligned}
f^\star\delta &= \delta, & f^\star\varpi &= \varpi\langle f^\star, f^\star\rangle,\\
\tau(f \times \delta) &= \delta, & \tau(f \times \varpi) &= \varpi\langle \tau(f \times \pi_1), \tau(f \times \pi_2)\rangle.
\end{aligned}
$$

The above definition forces the nondeterministic choice to be an *algebraic operation* in sense of [26]. This implies that the semilattice structure distributes over binding from the left (but not necessarily from the right) as reflected in our calculus in Section 3.

Remark 2. As in the original computational metalanguage [21], we work over an arbitrary base category $\mathbf{C}$, with $\mathbf{Set}$ and $\omega\mathbf{Cpo}$ as prominent instances, thus establishing our corecursion scheme at a high level of generality. Our calculus will be sound and complete w.r.t. the whole class of possible instances of $\mathbf{C}$. Restricting, e.g., to order-theoretic models will, of course, preserve soundness, while completeness may break down due to particular properties of the base category becoming observable in the calculus. The FIX-logic of Crole and Pitts [6] is sound and complete w.r.t. certain order-theoretic models compatible with our models, in particular with $\mathbf{Set}$.

Example 3. [21] The core examples of strong semi-additive monads are the finite powerset monad $\mathcal{P}_\omega$, or, in the domain-theoretic setting, various powerdomain constructions. Moreover, the powerset monad $\mathcal{P}$ and more generally, the *quantale monad* [16] $\lambda X.Q^X$ for a quantale [27] Q are strong semi-additive monads. Further examples of semi-additive monads can be obtained from basic ones by combining them with other effects, e.g. by adding probabilistic choice [32] or by applying suitable monad transformers. In particular, the following monad transformers (which produce a new monad Q, given a monad T) preserve semi-additivity over any base category with sufficient structure:

1. Exceptions: $QA = T(A + E)$,
2. States: $QA = S \to T(A \times S)$,
3. I/O: $QA = \mu X.\, T(A + I \to (X \times O))$,
4. Continuations: $QA = (A \to TK) \to TK$.

E.g., the non-deterministic state monad $TX = S \to P(S \times X)$, is a strong semi-additive monad both over $\mathbf{Set}$ (with P denoting any variant of powerset) and over any reasonable category of domains (with P denoting a powerdomain construction with deadlock).

To model processes which are composed of atomic steps to be thought of as pieces of imperative code with generic side-effects, we use a variant of the *resumption monad*

transformer [4]: Assuming that for every $X \in \mathrm{Ob}(\mathbf{C})$ the endofunctor $T(Id + X)$: $\mathbf{C} \to \mathbf{C}$ possesses a final coalgebra, which we denote by $\nu\gamma.\, T(\gamma + X)$, we define a new monad R by

$$RX = \nu\gamma.\, T(\gamma + X)$$

— R exists, e.g., if the base category is locally presentable and T is accessible [36], a basic example being $TX = S \to P(S \times X)$ where P is finite powerset or a powerdomain. Intuitively, a *resumption*, i.e. a computation in RX, takes an atomic step in T and then returns either a value in X or a further computation in RX, possibly continuing in this way indefinitely. Using a final coalgebra semantics amounts to identifying processes up to coalgebraic behavioural equivalence, which generalizes strong bisimilarity.

3 A Calculus for Side-effecting Processes

As originally observed by Moggi [21], strong monads support a *computational metalanguage*, i.e. essentially a generic sequential imperative programming language. Here we introduce a concurrent version of the metalanguage, the *concurrent metalanguage*, based semantically on the resumption monad transformer.

The concurrent metalanguage is parametrised over a countable signature Σ including a set of atomic types W, from which the type system is generated by the grammar

$$P ::= W \mid 1 \mid P \times P \mid P + P \mid TP \mid T_\nu P$$

— that is, we support sums and products, but not functional types, our main target being the common imperative programming basis, which does not include functional abstraction. Base effects are represented by T, and resumptions by T_ν.

Moreover, Σ includes function symbols $f : A \to B$ with given profiles, where A and B are types. The terms of the language, also referred to as *programs*, and their types are then determined by the rules shown in Fig. 1; the dotted line separates operators for sequential non-determinism from the process operators. Besides the standard term language for sums and products and the bind and return operators do and ret of the computational metalanguage, the concurrent metalanguage includes operations $\varnothing$ and $+$ are called *deadlock* and *choice*, respectively, as well as two specific constructs (*out* and *unfold*) for resumptions, explained later. Judgements $\Gamma \triangleright t : A$ read 'term t has type A in context Γ', where a *context* is a list $\Gamma = (x_1 : A_1, \dots, x_n : A_n)$ of typed variables. Programs whose type is of the form $T_\nu A$ are called *processes*. The notions of free and bound variables are defined in a standard way, as well as a notion of capture-avoiding substitution.

The semantics of the concurrent metalanguage is defined over ME_ν-*models*, referred to just as *models* below. A model is based on a distributive category [5] $\mathbf{C}$, i.e. a category with binary sums and finite products such that the canonical map $A \times B + A \times C \to A \times (B + C)$ is an isomorphism, with inverse $dist : A \times (B + C) \to A \times B + A \times C$ (this holds, e.g., when $\mathbf{C}$ is Cartesian closed). Moreover, it specifies a strong semi-additive monad T on $\mathbf{C}$ such that for every $A \in \mathrm{Ob}(\mathbf{C})$ the functor $T(Id + A)$ possesses a final coalgebra denoted $RA = \nu\gamma.\, T(\gamma + A)$, thus defining a functor R (*resumptions*).

$$
\textbf{(var)}\ \frac{x : A \in \Gamma}{\Gamma \rhd x : A} \qquad \textbf{(app)}\ \frac{f : A \to B \in \Sigma \quad \Gamma \rhd t : A}{\Gamma \rhd f(t) : B} \qquad \textbf{(1)}\ \frac{}{\Gamma \rhd \star : 1}
$$

$$
\textbf{(pair)}\ \frac{\Gamma \rhd t : A \quad \Gamma \rhd u : B}{\Gamma \rhd \langle t, u\rangle : A \times B} \qquad \textbf{(fst)}\ \frac{\Gamma \rhd t : A \times B}{\Gamma \rhd \mathsf{fst}\, t : A} \qquad \textbf{(snd)}\ \frac{\Gamma \rhd t : A \times B}{\Gamma \rhd \mathsf{snd}\, t : B}
$$

$$
\textbf{(case)}\ \frac{\Gamma \rhd s : A + B \quad \Gamma, x : A \rhd t : C \quad \Gamma, y : B \rhd u : C}{\Gamma \rhd \mathsf{case}\ s\ \mathsf{of}\ \mathsf{inl}\, x \mapsto t; \mathsf{inr}\, y \mapsto u : C} \qquad \textbf{(nil)}\ \frac{}{\Gamma \rhd \varnothing : TA}
$$

$$
\textbf{(inl)}\ \frac{\Gamma \rhd t : A}{\Gamma \rhd \mathsf{inl}\, t : A + B} \qquad \textbf{(inr)}\ \frac{\Gamma \rhd t : B}{\Gamma \rhd \mathsf{inr}\, t : A + B} \qquad \textbf{(ret)}\ \frac{\Gamma \rhd t : A}{\Gamma \rhd \mathsf{ret}\, t : TA}
$$

$$
\textbf{(do)}\ \frac{\Gamma \rhd p : TA \quad \Gamma, x : A \rhd q : TB}{\Gamma \rhd \mathsf{do}\ x \leftarrow p; q : TB} \qquad \textbf{(plus)}\ \frac{\Gamma \rhd p + q : TA}{\Gamma \rhd p : TA \quad \Gamma \rhd q : TA}
$$

$$
\textbf{(out)}\ \frac{\Gamma \rhd p : T_\nu A}{\Gamma \rhd \mathsf{out}(p) : T(T_\nu A + A)} \qquad \textbf{(unf)}\ \frac{\Gamma \rhd p : A \quad \Gamma, x : A \rhd q : T(A + B)}{\Gamma \rhd \mathsf{init}\, x := p\ \mathsf{unfold}\, \{q\} : T_\nu B}
$$

Fig. 1. Typing rules for the concurrent metalanguage

A model interprets base types as objects of $\mathbf{C}$. The interpretation $[\![A]\!]$ of types A is then defined by standard clauses for 1, $A \times B$, and $A + B$ and $[\![TA]\!] = T[\![A]\!]$, $[\![T_\nu A]\!] = R[\![A]\!]$. For $\Gamma = (x_1 : A_1, \dots, x_n : A_n)$ we put $[\![\Gamma]\!] = [\![A_1]\!] \times \dots \times [\![A_n]\!]$. Moreover, a model interprets function symbols $f : A \to B$ as morphisms $[\![f]\!] : [\![A]\!] \to [\![B]\!]$, which induces an interpretation $[\![t]\!] : [\![\Gamma]\!] \to [\![A]\!]$ of programs $\Gamma \rhd t : A$ given by the usual clauses for variables, function application, pairing, projections, injections, and $\star$. The operations $+$ and $\varnothing$ are interpreted by the bounded join semilattice operations ϖ and δ of T, respectively. For the monad operations and the case operator, we have

- $[\![\Gamma \rhd \mathsf{case}\ s\ \mathsf{of}\ \mathsf{inl}\, x \mapsto t; \mathsf{inr}\, y \mapsto u : C]\!] =$
  $\big[[\![\Gamma, x : A \rhd t : C]\!], [\![\Gamma, y : B \rhd u : C]\!]\big] \circ \mathit{dist} \circ \big\langle \mathrm{id}, [\![\Gamma \rhd s : A + B]\!]\big\rangle$,
- $[\![\Gamma \rhd \mathsf{do}\ x \leftarrow p; q : TB]\!] = [\![\Gamma, x : A \rhd q : TB]\!] \diamond \tau_{[\![\Gamma]\!],[\![A]\!]} \circ \langle \mathrm{id}, [\![\Gamma \rhd p : TA]\!]\rangle$,
- $[\![\Gamma \rhd \mathsf{ret}\, t : TA]\!] = \eta_A \circ [\![\Gamma \rhd t : A]\!]$,

where as usual $\langle f, g\rangle : A \to B \times C$ denotes pairing of morphisms $f : A \to B$, $g : A \to C$, and $[f, g] : A + B \to C$ denotes copairing of $f : A \to C$ and $g : B \to C$.

It remains to interpret out, which is just the final coalgebra structure of RA, and the loop construct $\mathsf{init}\, x := p\ \mathsf{unfold}\, \{q\}$ which captures coiteration. Formally, let $\alpha_A : RA \to T(RA + A)$ be the final coalgebra structure, and for a coalgebra $f : X \to T(X + A)$, let $\digamma_f : X \to RA$ be the unique coalgebra morphism. Then we put

$$
[\![\Gamma \rhd \mathsf{out}(p) : T(T_\nu A + A)]\!] = \alpha_{[\![A]\!]} \circ [\![\Gamma \rhd p : T_\nu A]\!]
$$
$$
[\![\Gamma \rhd \mathsf{init}\, x := p\ \mathsf{unfold}\, \{q\} : T_\nu B]\!] = R\pi_2 \circ \digamma_f \circ \langle \mathrm{id}, [\![\Gamma \rhd p : A]\!]\rangle
$$

$$
\begin{array}{ll}
\textbf{(case\_inl)} & \mathsf{case}\ \mathsf{inl}\,p\ \mathsf{of}\ \mathsf{inl}\,x \mapsto q;\ \mathsf{inr}\,y \mapsto r = q[p/x] \qquad \textbf{(fst)}\quad \mathsf{fst}\langle p,q\rangle = p \\
\textbf{(case\_inr)} & \mathsf{case}\ \mathsf{inr}\,p\ \mathsf{of}\ \mathsf{inl}\,x \mapsto q;\ \mathsf{inr}\,y \mapsto r = r[p/y] \qquad \textbf{(snd)}\quad \mathsf{snd}\langle p,q\rangle = q \\
\textbf{(case\_id)} & \mathsf{case}\ p\ \mathsf{of}\ \mathsf{inl}\,x \mapsto \mathsf{inl}\,x;\ \mathsf{inr}\,y \mapsto \mathsf{inr}\,y = p \qquad \textbf{(pair)}\quad \langle \mathsf{fst}\,p, \mathsf{snd}\,p\rangle = p \\
\textbf{(case\_sub)} & \begin{array}{l} \mathsf{case}\ p\ \mathsf{of}\ \mathsf{inl}\,x \mapsto t[q/z];\ \mathsf{inr}\,y \mapsto t[r/z] \\ \quad = t[\mathsf{case}\ p\ \mathsf{of}\ \mathsf{inl}\,x \mapsto q;\ \mathsf{inr}\,y \mapsto r/z] \end{array} \qquad (x,y \notin \mathrm{Vars}(r)) \\
\textbf{(}\star\textbf{)} & p : 1 = \star \qquad \textbf{(unit}_1\textbf{)}\quad \mathsf{do}\ x \leftarrow p; \mathsf{ret}\,x = p \qquad \textbf{(unit}_2\textbf{)}\quad \mathsf{do}\ x \leftarrow \mathsf{ret}\,a; p = p[a/x] \\
\textbf{(assoc)} & \mathsf{do}\ x \leftarrow (\mathsf{do}\ y \leftarrow p; q); r = \mathsf{do}\ x \leftarrow p; y \leftarrow q; r \qquad (y \notin \mathrm{Vars}(r))
\end{array}
$$

$$
\begin{array}{ll}
\textbf{(nil)}\quad p + \varnothing = p & \textbf{(comm)}\quad p + q = q + p \qquad \textbf{(idem)}\quad p + p = p \\
\textbf{(assoc\_plus)}\quad p + (q + r) = (p + q) + r & \textbf{(dist\_nil)}\quad \mathsf{do}\ x \leftarrow \varnothing; r = \varnothing \\
\textbf{(dist\_plus)}\quad \mathsf{do}\ x \leftarrow (p+q); r = \mathsf{do}\ x \leftarrow p; r + \mathsf{do}\ x \leftarrow q; r &
\end{array}
$$

$$
\textbf{(co-iter)}\quad \frac{\mathsf{out}(p) = \mathsf{next}\ q\ \mathsf{is}\ \mathsf{rest}\,x \mapsto \mathsf{cont}\,p;\ \mathsf{done}\,y \mapsto \mathsf{stop}\,y}{p[y/x] = \mathsf{init}\,x := y\ \mathsf{unfold}\,\{q\}}
$$

Fig. 2. Axiomatization of the concurrent metalanguage

where $f = T(dist) \circ \tau\langle \pi_1, g\rangle$ with $g = [\![\Gamma, x : A \rhd q : T(A + B)]\!]$. Thus, $\mathit{\Gamma}_f$ is uniquely determined by the commutative diagram

$$
\begin{array}{ccc}
[\![\Gamma]\!] \times [\![A]\!] & \xrightarrow{T(dist)\circ\tau\langle\pi_1, g\rangle} & T([\![\Gamma]\!] \times [\![A]\!] + [\![\Gamma]\!] \times [\![B]\!]) \\
\Big\downarrow{\scriptstyle \mathit{\Gamma}_f} & & \Big\downarrow{\scriptstyle T(\mathit{\Gamma}_f + \mathrm{id})} \\
R([\![\Gamma]\!] \times [\![B]\!]) & \xrightarrow{\alpha_{[\![\Gamma]\!] \times [\![B]\!]}} & T(R([\![\Gamma]\!] \times [\![B]\!]) + [\![\Gamma]\!] \times [\![B]\!]).
\end{array}
$$

A model is said to *satisfy* a well-typed equation $\Gamma \rhd t = s$ if $[\![\Gamma \rhd t : A]\!] = [\![\Gamma \rhd s : A]\!]$.

As suggestive abbreviations for use in process definitions, we write cont for $(\mathsf{ret}\,\mathsf{inl})$ and stop for $(\mathsf{ret}\,\mathsf{inr})$. Moreover, we write $(\mathsf{next}\ p\ \mathsf{is}\ \mathsf{rest}\,x \mapsto q;\ \mathsf{done}\,y \mapsto r)$ for $(\mathsf{do}\ z \leftarrow p; \mathsf{case}\ z\ \mathsf{of}\ \mathsf{inl}\,x \mapsto q;\ \mathsf{inr}\,y \mapsto r)$. We also define a converse $\mathsf{tuo} : T(T_\nu A + A) \to T_\nu A$ to out by

$$
\mathsf{tuo}(p) \;=\; \mathsf{init}\,q := p\ \mathsf{unfold}\,\{\mathsf{next}\ q\ \mathsf{is}\ \mathsf{rest}\,y \mapsto \mathsf{cont}(\mathsf{out}(y));\ \mathsf{done}\,x \mapsto \mathsf{stop}\,x\}.
$$

An axiomatization ME_ν of the concurrent metalanguage is given in Fig. 2 (where we omit the standard equational logic ingredients including the obvious congruence rules). Apart from the standard axioms for products and coproducts, ME_ν contains three well-known monad laws, axioms for semi-additivity (middle section) and a novel (bidirec-

tional) rule **(co-iter)** for effectful co-iteration (which in particular can be used to show that tuo is really inverse to out).

Theorem 4. *$\mathsf{ME}_{\boldsymbol{\nu}}$ is sound and strongly complete over $\mathsf{ME}_{\boldsymbol{\nu}}$-models.*

A core result on the concurrent metalanguage is an expressive corecursion scheme supported by the given simple axiomatisation. Its formulation requires n-ary coproducts $A_1 + \cdots + A_n$ with coproduct injections $\mathsf{inj}^n_i : A_i \to A_1 + \ldots + A_n$ and a correspondingly generalized case construct; all this can clearly be encoded in $\mathsf{ME}_{\boldsymbol{\nu}}$.

Theorem and Definition 5 (Mutual corecursion). *Let $f_i : A_i \to T_\nu B_i$, $i = 1, \ldots, k$ be fresh function symbols. A* guarded corecursive scheme *is a system of equations*

$$\mathsf{out}(f_i(x)) = \mathsf{do}\ z \leftarrow p_i; \mathsf{case}\ z\ \mathsf{of}\ \mathsf{inj}^{n_i}_1\, x_1 \mapsto p^i_1;\ \ldots\ ; \mathsf{inj}^{n_i}_{n_i}\, x_{n_i} \mapsto p^i_{n_i}$$

for $i = 1, \ldots, k$ such that for every i, p_i does not contain any f_j, and for every i, j p^i_j either does not contain any f_m or is of the form $p^i_j \equiv \mathsf{cont}\, f_m(x_j)$ for some m. Such a guarded corecursive scheme uniquely defines $f_1, \ldots, f_k$ (as morphisms in the model), and the solutions f_i are expressible as programs in $\mathsf{ME}_{\boldsymbol{\nu}}$.

As a first application of guarded corecursive schemes, we define a binding operation do_ν with the same typing as do but with T replaced by T_ν corecursively by

$$\mathsf{out}(\mathsf{do}_\nu\ x \leftarrow p; q) = \mathsf{next}\ \mathsf{out}(p)\ \mathsf{is}\ \mathsf{rest}\, x\ \mapsto \mathsf{cont}(\mathsf{do}_\nu\ x \leftarrow p; q);\ \mathsf{done}\, x \mapsto \mathsf{out}(q).$$

Similarly, we define operations ret_ν, $\varnothing_\nu$, and $+_\nu$ as analogues of ret, $\varnothing$, and $+$ by putting $\mathsf{ret}_\nu\, p = \mathsf{tuo}(\mathsf{stop}\, p)$, $\varnothing_\nu = \mathsf{tuo}(\varnothing)$, and $p +_\nu q = \mathsf{tuo}(\mathsf{out}(p) + \mathsf{out}(q))$. *These operations turn T_ν into a strong-semiadditive monad*; formally, we can derive (in $\mathsf{ME}_{\boldsymbol{\nu}}$) the top and middle sections of Fig. 2 with T replaced by T_ν (the monad laws already follow from results of [34]).

4 Programming with Side-effecting Processes

Above, we have begun to define operations on processes; in particular, $\varnothing_\nu$ is a deadlocked process, and $+_\nu$ is a nondeterministic choice of two processes. We next show how to define more complex operations, including parallel composition, by means of guarded corecursive schemes.

Note that over distributive categories, one can define the type of Booleans with the usual structure as $2 = 1 + 1$. We write $(\mathsf{if}\, b\, \mathsf{then}\, p\, \mathsf{else}\, q)$ as an abbreviation for $(\mathsf{case}\ b\ \mathsf{of}\ \mathsf{inl}\, x \mapsto p;\ \mathsf{inr}\, x \mapsto q)$ where b has type 2.

Sequential composition Although T_ν is a monad, its binding operator is not quite what one would want as sequential composition of processes, as it merges the last step of the first process with the first step of the second process. We can, however, capture sequential composition (with the same typing) in the intended way by putting

$$\mathsf{seq}\ x \leftarrow p; q = \mathsf{do}_\nu\ x \leftarrow p; \mathsf{tuo}(\mathsf{cont}\, q).$$

Branching Using the effect-free if operator defined earlier, we can define a conditional branching operator for processes $p, q : T_\nu A$ and a condition $b : T2$ by

$$\mathsf{if}_\nu\; b \;\mathsf{then}\; p \;\mathsf{else}\; q = \mathsf{tuo}\big(\mathsf{do}\; z \leftarrow b; \mathsf{if}\; z \;\mathsf{then}\; (\mathsf{cont}\; p) \;\mathsf{else}\; (\mathsf{cont}\; q)\big).$$

Looping For terms $\Gamma \rhd p$; $\Gamma, x : A \rhd b : T2$; and $\Gamma, x : A \rhd q : T_\nu A$, we define loops

$$\Gamma \rhd \mathsf{init}\; x := p \;\mathsf{while}\; b \;\mathsf{do}\; q : T_\nu A \qquad \text{and} \qquad \Gamma \rhd \mathsf{init}\; x := p \;\mathsf{do}\; q \;\mathsf{until}\; b : T_\nu A$$

as follows. We generalize the until loop to a program $U^b_{x,q}(r)$ for $\Gamma \rhd r : T_\nu A$ intended to represent $\mathsf{seq}\; y \leftarrow r; \mathsf{init}\; x := y \;\mathsf{while}\; b \;\mathsf{do}\; q$ (so that $(\mathsf{init}\; x := p \;\mathsf{do}\; q \;\mathsf{until}\; b) = U^b_{x,q}(q[p/x])$) and abbreviate $W^b_{x,q}(p) = (\mathsf{init}\; x := p \;\mathsf{while}\; b \;\mathsf{do}\; q)$. We then define the four functions $W^b_{x,q}$ and $U^b_{x,q}$ (for $b : 2$) by the guarded corecursive scheme

$$\mathsf{out}(W^b_{x,q}(p)) = \mathsf{do}\; v \leftarrow b[p/x]; \mathsf{if}\; v \;\mathsf{then}\; \mathsf{cont}(U^{\neg b}_{x,q}(q[p/x])) \;\mathsf{else}\; \mathsf{stop}(p),$$
$$\mathsf{out}(U^b_{x,q}(r)) = \mathsf{next}\; \mathsf{out}(r) \;\mathsf{is}\; \mathsf{rest}\; z \mapsto \mathsf{cont}(U^b_{x,q}(z)); \mathsf{done}\; y \mapsto \mathsf{cont}(W^{\neg b}_{x,q}(y)).$$

Exceptions As the concurrent metalanguage includes coproducts, the exception monad transformer($T^E A = T(A + E)$) [4] and the corresponding operations for raising and handling exceptions are directly expressible in ME_ν.

Interleaving We introduce process interleaving $\parallel : T_\nu A \times T_\nu B \to T_\nu(A \times B)$ by a CCS-style expansion law [20] (using an auxiliary left merge $\lfloor\!\lfloor$)

$$\begin{aligned}
\mathsf{out}(p \parallel q) &= \mathsf{out}(p \lfloor\!\lfloor q) + \mathsf{do}\; \langle x, y\rangle \leftarrow \mathsf{out}(q \lfloor\!\lfloor p); \mathsf{ret}\langle y, x\rangle,\\
\mathsf{out}(p \lfloor\!\lfloor q) &= \mathsf{next}\; \mathsf{out}(p) \;\mathsf{is}\; \mathsf{rest}\; r \mapsto \mathsf{cont}(r \parallel q);\\
&\qquad\qquad\qquad\qquad \mathsf{done}\; x \mapsto \mathsf{cont}(\mathsf{do}_\nu\; y \leftarrow q; \mathsf{ret}_\nu\langle x, y\rangle).
\end{aligned}$$

This is easily seen to be equivalent to the guarded corecursive scheme

$$\begin{aligned}
\mathsf{out}(p \parallel q) = \;& \mathsf{do}\; u \leftarrow (p \lfloor q + p \rfloor q);\\
& \mathsf{case}\; u \;\mathsf{of}\; \mathsf{inl}\langle s, t\rangle \mapsto \mathsf{cont}(s \parallel t); \mathsf{inr}\; r \mapsto \mathsf{cont}\; r
\end{aligned}$$

where for $p : T_\nu A$, $q : T_\nu B$, $p \lfloor q : T(T_\nu A \times T_\nu B + T_\nu(A \times B))$ is defined as

$$p \lfloor q = \mathsf{next}\; \mathsf{out}(p) \;\mathsf{is}\; \mathsf{rest}\; r \mapsto \mathsf{ret}\; \mathsf{inl}\langle r, q\rangle; \mathsf{done}\; x \mapsto \mathsf{ret}\; \mathsf{inr}(\mathsf{do}_\nu\; y \leftarrow q; \mathsf{ret}_\nu\langle x, y\rangle)$$

and $p \rfloor q : T(T_\nu A \times T_\nu B + T_\nu(A \times B))$ is the evident dual of $p \lfloor q$.

5 Verification and Process Invariants

We now explore the potential of our formalism as a *verification* framework, extending existing monad-based program logics [29, 30] to concurrent processes. A cornerstone of these frameworks is a notion of *pure* program:

Definition 6 (Pure programs). A program $p : TA$ is *pure* if

– p is *discardable*, i.e., $\mathsf{do}\ y \leftarrow p; \mathsf{ret}\,\star = \mathsf{ret}\,\star$;
– p is *copyable*, i.e. $\mathsf{do}\ x \leftarrow p; y \leftarrow p; \mathsf{ret}\langle x, y\rangle = \mathsf{do}\ x \leftarrow p; \mathsf{ret}\langle x, x\rangle$; and
– p commutes with any other discardable and copyable program q, i.e.
$(\mathsf{do}\ x \leftarrow p; y \leftarrow q; \mathsf{ret}\langle x, y\rangle) = \mathsf{do}\ y \leftarrow q; x \leftarrow p; \mathsf{ret}\langle x, y\rangle$.

Intuitively, pure programs are those that can access internal data behind the computation but cannot affect it. A typical example of a pure program is a getter method. As shown in [29], pure programs form a submonad P of T. A *test* is a program of type $P2$. All logical connectives extend to tests; e.g. $\neg b = (\mathsf{do}\ x \leftarrow b; \mathsf{ret}\,\neg x)$ for $b : P2$.

Given a program $p : TA$ and tests $\phi, \psi : P2$, the program $\mathsf{filter}(p, \phi, \psi) : TA$ is defined by the equation

$$\mathsf{filter}(p, \phi, \psi) = \mathsf{do}\ x \leftarrow \phi; y \leftarrow p; z \leftarrow \psi; \mathsf{if}\,(x \Rightarrow z)\ \mathsf{then\ ret}\,y\ \mathsf{else}\ \varnothing.$$

Intuitively, filter modifies the given program p by removing those threads that satisfy the precondition ϕ but fail the postcondition ψ. This enables us to encode a Hoare triple (alternatively to [29, 30]) by the equivalence

$$\{\phi\}p\{\psi\} \iff \mathsf{filter}(p, \phi, \psi) = p$$

— i.e. the Hoare triple $\{\phi\}p\{\psi\}$ is satisfied iff $\mathsf{filter}(p, \phi, \psi)$ does not remove any execution paths from p. On the other hand, filter extends to processes as follows:

$$\mathsf{filter}_\nu(p, \phi, \psi) = \mathsf{init}\,z := p\ \mathsf{unfold}\,\{\mathsf{tuo}(\mathsf{filter}(\mathsf{out}(z), \phi, \psi))\}.$$

It turns out that the above definition of Hoare triple is equivalent to the one from [29, 30], which in particular enables use of the sequential monad-based Hoare calculus of [30]:

Lemma 7. *For every program p and tests ϕ, ψ, $\{\phi\}p\{\psi\}$ is equivalent to the equation*

$$\begin{aligned}&\mathsf{do}\ x \leftarrow \phi; y \leftarrow p; z \leftarrow \psi; \mathsf{ret}\langle x, y, z, x \Rightarrow z\rangle = \\ &\mathsf{do}\ x \leftarrow \phi; y \leftarrow p; z \leftarrow \psi; \mathsf{ret}\langle x, y, z, \top\rangle.\end{aligned}$$

A test ϕ is an *invariant* of process p if $\mathsf{filter}_\nu(p, \phi, \phi) = p$. We use $\mathsf{inv}(p, \phi)$ as a shorthand for this equality. Given a process $p : T_\nu A$, we define *partial execution* of p by

$$\mathsf{exec}(p) = \mathsf{tuo}(\mathsf{next}\ p\ \mathsf{is\ rest}\,x \mapsto \mathsf{out}(x);\ \mathsf{done}\,x \mapsto \mathsf{stop}\,x).$$

For every p, $\mathsf{exec}(p)$ is precisely the program obtained by collapsing the first and the second steps of p into one. We denote by $\mathsf{exec}^n(p)$ the n-fold application of exec to p. This allows us to formalize satisfaction of a safety property ϕ by a process p:

‘p is safe w.r.t. ψ at ϕ’ iff for every n, $\{\phi\}\ \mathsf{exec}^n(p)\{\psi\}$.

Note, however, that this definition is not directly expressible in our logic, because it involves quantification over the naturals. Often this problem can be overcome by picking out a suitable process invariant.

Lemma 8. *Let ϕ, ψ, and ξ be tests such that $\phi \Rightarrow \xi$ and $\xi \Rightarrow \psi$, and let p be a process. Then $\mathsf{inv}(p, \xi)$ implies $\{\phi\}\ \mathsf{exec}^n(p)\{\psi\}$ for every n.*

6 Worked Example: Dekker's Mutual Exclusion Algorithm

We illustrate the use of our calculus by encoding Dekker's mutual exclusion algorithm. This algorithm was originally presented as an Algol program, and hence presumes some fixed imperative semantics, while we present (and verify) a version with *generic* side-effects. We introduce the following signature symbols:

$$\begin{aligned} &\mathrm{set\_flag} : 2 \times 2 \to T1, \qquad &&\mathrm{set\_turn} : 2 \to T1,\\ &\mathrm{get\_flag} : 2 \to P2, &&\mathrm{turn\_is} : 2 \to P2, \end{aligned}$$

which can be roughly understood as interface functions accessing variables `flag1`, `flag2` and `turn`. This is justified by a suitable equational axiomatization of the above operators, which includes the following axioms (we assume $i \neq j$):

$$\begin{aligned} \mathsf{do}\ \mathrm{set\_flag}(i,b); \mathrm{get\_flag}(i) &= \mathsf{do}\ \mathrm{set\_flag}(i,b); \mathsf{ret}\, b\\ \mathsf{do}\ \mathrm{set\_flag}(i,b); \mathrm{set\_flag}(j,c) &= \mathsf{do}\ \mathrm{set\_flag}(j,c); \mathrm{set\_flag}(i,b)\\ \mathsf{do}\ \mathrm{set\_flag}(i,b); \mathrm{set\_flag}(i,c) &= \mathrm{set\_flag}(i,c) \end{aligned}$$

$$\begin{aligned} \mathsf{do}\ \mathrm{set\_turn}(b); \mathrm{turn\_is}(c) &= \mathsf{do}\ \mathrm{set\_turn}(b); \mathsf{ret}(b \Leftrightarrow c)\\ \mathsf{do}\ \mathrm{set\_turn}(b); \mathrm{set\_turn}(c) &= \mathrm{set\_turn}(c). \end{aligned}$$

(Obvious further axioms are omitted.) The crucial part of Dekker's algorithm is a (sub)program implementing *busy waiting*. In our case this is captured by the function $busy\_wait : 2 \to T1$ defined as follows:

$$\begin{aligned} busy\_wait(i) = \mathsf{while}\ &\mathrm{get\_flag}(\mathrm{flip}(i))\ \mathsf{do}\ \mathsf{if}_\nu\ \mathrm{turn\_is}(\mathrm{flip}(i))\\ &\mathsf{then}\ \mathsf{seq}\,[\mathrm{set\_flag}(i,\bot)]; \mathsf{await}(\mathrm{turn\_is}(i))\\ &\mathsf{else}\ [\mathrm{set\_flag}(i,\top)] \end{aligned}$$

Here, we used the following shorthands: $[p] = \mathsf{tuo}(\mathsf{stop}\, p)$ denotes the one-step process defined by $p : TA$, $\mathrm{flip} : 2 \to 2$ is the function swapping the coproduct components of $2 = 1 + 1$; $(\mathsf{while}\ b\ \mathsf{do}\ q)$ encodes $(\mathsf{init}\, x \leftarrow \star\ \mathsf{while}\ b\ \mathsf{do}\ q)$; finally, $(\mathsf{await}\, b)$ with b of type $T2$, intuitively meaning 'wait until b', is defined by the equation:

$$\mathsf{await}\, b = \mathsf{while}\ \neg b\ \mathsf{do}\ \mathsf{ret}_\nu \star.$$

Finally, we define a generic process accessing the critical section:

$$\begin{aligned} proc(i,p) = \mathsf{seq}\, &[\mathrm{set\_flag}(i,\top)]; busy\_wait(i);\\ &[\mathrm{in\_cs}(i)]; p; [\mathrm{out\_cs}(i)];\\ &[\mathrm{set\_turn}(\mathrm{flip}(i))]; [\mathrm{set\_flag}(i,\bot)]. \end{aligned}$$

Here we use the functions $\mathrm{in\_cs}, \mathrm{out\_cs} : 2 \to T1$ in order to keep track of the beginning and the end of the critical section. These functions are supposed to work together with the testing function $\mathrm{cs} : 2 \to T2$ as prescribed by the axioms

$$\begin{aligned} \mathsf{do}\ \mathrm{in\_cs}(i); \mathrm{cs}(i) &= \mathsf{do}\ \mathrm{in\_cs}(i); \mathsf{ret}\,\top,\\ \mathsf{do}\ \mathrm{out\_cs}(i); \mathrm{cs}(i) &= \mathsf{do}\ \mathrm{out\_cs}(i); \mathsf{ret}\,\bot. \end{aligned}$$

Now the safety condition for the algorithm can be expressed by the formula

$$\forall n.\, \{\neg\mathrm{cs}(\bar{1}) \wedge \neg\mathrm{cs}(\bar{2})\}\, \mathsf{exec}^n(proc(\bar{1}, p) \parallel proc(\bar{2}, q))\{\neg\mathrm{cs}(\bar{1}) \vee \neg\mathrm{cs}(\bar{2})\}$$

where $\bar{1}$ and $\bar{2}$ are the canonical coproduct injections $\mathsf{inl}\,\star$ and $\mathsf{inr}\,\star$. By Lemma 8, it suffices to show that the following formula

$$\neg\mathrm{cs}(\bar{1}) \wedge \mathrm{cs}(\bar{2}) \wedge \mathrm{get\_flag}(\bar{2})\ \vee \neg\mathrm{cs}(\bar{2}) \wedge \mathrm{cs}(\bar{1}) \wedge \mathrm{get\_flag}(\bar{1})\ \vee \neg\mathrm{cs}(\bar{1}) \wedge \neg\mathrm{cs}(\bar{2})$$

is an invariant of $proc(\bar{1}, p) \parallel proc(\bar{2}, q)$. As can be shown by definition of parallel composition, this holds iff the same formula is an invariant of both $proc(\bar{1}, p)$ and $proc(\bar{2}, q)$, which in turn can be shown by coinduction in ME_ν.

7 Conclusions and further work

We have studied asynchronous concurrency in a framework of generic effects. To this end, we have combined the theories of computational monads and final coalgebras to obtain a framework that generalizes process algebra to encompass processes with side-effecting steps. We have presented a sound and complete equational calculus for the arising *concurrent metalanguage* ME_ν, and we have obtained a syntactic corecursion scheme in which corecursive functions are syntactically reducible to a basic loop construct. Within this calculus, we have given generic definitions for standard imperative constructs and a number of standard process operators, most notably parallel composition.

Although the proof principles developed so far are already quite powerful, as was shown in an example verification of a generic mutual exclusion scheme following Dekker's algorithm, we intend to develop more expressive verification logics for side-effecting processes, detached from equational reasoning. Initial results of this kind have already been used in the example verification, specifically an encoding of generic Hoare triples and an associated proof principle for safety properties. An interesting perspective in this direction is to identify a variant of the assume/guarantee principle for side-effecting processes (cf. e.g. [7]). A further topic of investigation is to develop weak notions of process equivalence in our framework, such as testing equivalence [22].

Finally, the decidability status of ME_ν remains open. Note that in case of a positive answer, all equations between functions defined by corecursion schemes, e.g. process algebra identities, become decidable. While experience suggests that even very simple calculi that combine loop constructs with monadic effects tend to be undecidable, the corecursion axiom as a potential source of trouble seems rather modest, and no evident encoding of an undecidable problem appears to be directly applicable.

References

[1] F. Bartels. Generalised coinduction. *Math. Structures in Comp. Sci.*, 13(2):321–348, 2003.

[2] N. Benton and M. Hyland. Traced premonoidal categories. *ITA*, 37(4):273–299, 2003.

[3] V. Capretta. General recursion via coinductive types. *Logical Methods in Computer Science*, 1(2), 2005.

[4] P. Cenciarelli and E. Moggi. A syntactic approach to modularity in denotational semantics. Technical report, Category Theory and Computer Science, 1993.

[5] J. R. B. Cockett. Introduction to distributive categories. *Mathematical Structures in Computer Science*, 3(3):277–307, 1993.

[6] R. L. Crole and A. M. Pitts. New foundations for fixpoint computations. In *Logic in Computer Science, LICS 1990*, pp. 489–497. IEEE, 1990.

[7] W. P. de Roever, F. S. de Boer, U. Hannemann, J. Hooman, Y. Lakhnech, M. Poel, and J. Zwiers. *Concurrency Verification: Introduction to Compositional and Noncompositional Methods*. Cambridge University Press, 2001.

[8] L. Erkök and J. Launchbury. Recursive monadic bindings. In *ICFP'00*, pp. 174–185. ACM, 2000.

[9] A. Filinski. On the relations between monadic semantics. *Theor. Comp. Sci.*, 375:41–75, 2007.

[10] M. P. Fiore, E. Moggi, and D. Sangiorgi. A fully abstract model for the π-calculus. *Inf. Comput.*, 179(1):76–117, 2002.

[11] S. Goncharov. *Kleene monads*. PhD thesis, Universität Bremen, 2010.

[12] P. Hancock and A. Setzer. Guarded induction and weakly final coalgebras in dependent type theory. In *From Sets and Types to Topology and Analysis. Towards Practicable Foundations for Constructive Mathematics*, pp. 115 – 134, 2005.

[13] W. L. Harrison. The essence of multitasking. In *AMAST'06*, vol. 4019 of *LNCS*, pp. 158–172. Springer, 2006.

[14] W. L. Harrison and J. Hook. Achieving information flow security through monadic control of effects. *J. Computer Security*, 17:599–653, 2009.

[15] M. Hennessy and G. D. Plotkin. Full abstraction for a simple parallel programming language. In *MFCS*, pp. 108–120, 1979.

[16] B. Jacobs. From coalgebraic to monoidal traces. *Electron. Notes Theor. Comput. Sci.*, 264(2):125–140, 2010.

[17] B. Jacobs and E. Poll. Coalgebras and Monads in the Semantics of Java. *Theoret. Comput. Sci.*, 291:329–349, 2003.

[18] S. Krstic, J. Launchbury, and D. Pavlovic. Categories of processes enriched in final coalgebras. In *FOSSACS'01*, vol. 2030 of *LNCS*, pp. 303–317. Springer, 2001.

[19] S. Milius, T. Palm, and D. Schwencke. Complete iterativity for algebras with effects. In *CALCO'09*, pp. 34–48. Springer-Verlag, 2009.

[20] R. Milner. *Communication and concurrency*. Prentice-Hall, Inc., 1989.

[21] E. Moggi. Notions of computation and monads. *Inf. Comput.*, 93:55–92, 1991.

[22] R. D. Nicola and M. Hennessy. Testing equivalence for processes. In *International Colloquium on Automata, Languages and Programming, ICALP 1983*, vol. 154 of *LNCS*, pp. 548–560. Springer, 1983.

[23] N. Papaspyrou and D. Macos. A study of evaluation order semantics in expressions with side effects. *J. Funct. Program.*, 10(3):227–244, 2000.

[24] N. S. Papaspyrou. A resumption monad transformer and its applications in the semantics of concurrency. In *3rd Panhellenic Logic Symposium, Anogia*, 2001.

[25] S. Peyton Jones, editor. *Haskell 98 Language and Libraries — The Revised Report*. Cambridge, 2003. also: *J. Funct. Programming* 13, 2003.

[26] G. Plotkin and J. Power. Semantics for algebraic operations. In *Mathematical Foundations of Programming Semantics, MFPS 2001*, vol. 45 of *ENTCS*, 2001.

[27] K. I. Rosenthal. *Quantales and their applications*. Number 234 in Pitman Research Notes in Mathematics Series. Longman Scientific & Technical, 1990.

[28] J. Rutten. Universal coalgebra: A theory of systems. *Theoret. Comput. Sci.*, 249:3–80, 2000.

[29] L. Schröder and T. Mossakowski. Monad-independent dynamic logic in HasCASL. *J. Logic Comput.*, 14:571–619, 2004.

[30] L. Schröder and T. Mossakowski. HasCASL: Integrated higher-order specification and program development. *Theor. Comput. Sci.*, 410(12-13):1217–1260, 2009.

[31] D. Syme, A. Granicz, and A. Cisternino. *Expert F#*. Apress, 2007.

[32] R. Tix, K. Keimel, and G. Plotkin. Semantic domains for combining probability and non-determinism. *ENTCS*, 222:3–99, 2009.

[33] A. P. Tolmach and S. Antoy. A monadic semantics for core Curry. In *WFLP'03*, vol. 86(3) of *ENTCS*, pp. 16–34, 2003.

[34] T. Uustalu. Generalizing substitution. *ITA*, 37(4):315–336, 2003.

[35] P. Wadler. How to declare an imperative. *ACM Computing Surveys*, 29:240–263, 1997.

[36] J. Worrell. Terminal sequences for accessible endofunctors. In *Coalgebraic Methods in Computer Science, CMCS 99*, vol. 19 of *ENTCS*, 1999.

Appendix: Proof details.

We justify the definition of the injections $\mathsf{inj}_i^n : A_i \to A_1 + \dots + A_n$ by the recursive equations:

$$\mathsf{inj}_1^1\, p = p, \qquad \mathsf{inj}_1^{n+1}\, p = \mathsf{inl}\, p, \qquad \mathsf{inj}_{i+1}^{n+1}\, p = \mathsf{inr}\, \mathsf{inj}_i^n\, p.$$

We call a *guarded corecursive equation* an equation of the form

$$\mathsf{out}(f(x)) = \mathsf{do}\ z \leftarrow p; \mathsf{case}\ z\ \mathsf{of}\ \mathsf{inj}_1^n\, x_1 \mapsto p_1; \dots; \mathsf{inj}_n^n\, x_n \mapsto p_n \qquad (**)$$

if f does not occur in p and none of the p_i contains f unless $p_i \equiv \mathsf{cont}\, f(x_i)$ (which is, of course, well-typed only in case $A_i = A$). The fact that the right-hand side is prefixed with the binding $z \leftarrow p$ plays an important role for the expressiveness of the scheme; a comparatively trivial point in this respect is that this allows substituting the arguments x_i in $\mathsf{cont}\, f(x_i)$ by arbitrary terms.

Lemma 9 (Corecursion). *Given some appropriately typed programs p and the p_i such that (**) is guarded, there is a unique function f satisfying (**) and this function is defined by an effectively computable metalanguage term.*

Proof. The idea is to start from a special case and successively extend generality.

(i) Suppose that $n = 2$, $p_1 \equiv \mathsf{cont}\, f(x)$ and p_2 does not contain f. Let $A \to T_\nu B$ be the type profile of f and let us define a function $F : A + T_\nu B \to T_\nu B$ by putting: $F(z) = \big(\mathsf{init}\ z := z\ \mathsf{unfold}\, \{H(z)\}\big)$ where

$$\begin{aligned} H(z) = \mathsf{case}\ z\ \mathsf{of}\ \mathsf{inl}\, x \mapsto \big(&\mathsf{do}\ z \leftarrow p; \mathsf{case}\ z\ \mathsf{of} \\ &\mathsf{inl}\, x_1 \mapsto \mathsf{cont}(\mathsf{inl}\, x_1); \\ &\mathsf{inr}\, x_2 \mapsto (\, \mathsf{next}\ p_2\ \mathsf{is}\ \mathsf{rest}\, x\ \mapsto \mathsf{cont}(\mathsf{inr}\, x); \\ &\qquad\qquad\qquad\qquad \mathsf{done}\, x\ \mapsto \mathsf{stop}\, x)\big); \\ \mathsf{inr}\, r \mapsto \big(&\mathsf{next}\ \mathsf{out}(r)\ \mathsf{is}\ \mathsf{rest}\, x\ \mapsto \mathsf{cont}(\mathsf{inr}\, x); \\ &\qquad\qquad \mathsf{done}\, x\ \mapsto \mathsf{stop}\, x\big). \end{aligned}$$

Let us show that $F(\mathsf{inr}\, x) = x$. By **(corec)**,

$$\mathsf{out}(F(z)) = \mathsf{next}\ H(z)\ \mathsf{is}\ \mathsf{rest}\, x \mapsto \mathsf{cont}\, F(z); \mathsf{done} \mapsto \mathsf{stop}\, x. \qquad (1)$$

Note that

$$H(\mathsf{inr}\, x) = \mathsf{next}\ \mathsf{out}(x)\ \mathsf{is}\ \mathsf{rest}\, x \mapsto \mathsf{cont}(\mathsf{inr}\, x); \mathsf{done}\, x \mapsto \mathsf{stop}\, x$$

from which we conclude that

$$\mathsf{out}(F(\mathsf{inr}\, x)) = \mathsf{next}\ \mathsf{out}(x)\ \mathsf{is}\ \mathsf{rest}\, x \mapsto \mathsf{cont}\, F(\mathsf{inr}\, x); \mathsf{done}\, x \mapsto \mathsf{stop}\, x.$$

The latter means that $(F \circ \mathsf{inr})$ satisfies the same equation as the identity function. Hence, by **(corec)** both these functions must be provably equal, i.e. for every x, $F(\mathsf{inr}\,x) = x$. Now, (1) can be simplified down to:

$$\begin{aligned}\mathsf{out}(F(z)) = \mathsf{case}\ z\ \mathsf{of}\ \mathsf{inl}\,x\ &\mapsto \big(\mathsf{next}\ p\ \mathsf{is}\ \mathsf{rest}\,x_1\ \mapsto \mathsf{cont}\,F(\mathsf{inl}(x_1));\\ &\qquad\qquad\qquad\qquad \mathsf{done}\,x_2\ \mapsto p_2\big);\\ \mathsf{inr}\,r\ &\mapsto r.\end{aligned}$$

By **(corec)** F is uniquely defined by this equation. It can be verified by routine calculations that $f(x) = F(\mathsf{inl}\,x)$ is a solution of ($**$). In order to prove uniqueness, let us assume that g is some other solution of ($**$). Let

$$G(z) = \mathsf{case}\ z\ \mathsf{of}\ \mathsf{inl}\,x \mapsto g(x);\ \mathsf{inr}\,r \mapsto \mathsf{tuo}(r).$$

Clearly, $g(x) = G(\mathsf{inl}\,x)$ and it can be shown that G satisfies the equation defining F. Therefore $g(x) = G(\mathsf{inl}\,x) = F(\mathsf{inl}\,x) = f(x)$ and we are done.

(ii) Let $n > 1$, $p_1 \equiv \mathsf{cont}\,f(x_1)$ and for every $i > 1$, p_i does not contain f. We reduce this case to the previous one as follows. Observe that, by assumption,

$$\mathsf{out}(f(x)) = \mathsf{do}\ z \leftarrow p; \mathsf{case}\ z\ \mathsf{of}\ \mathsf{inl}\,x_1 \mapsto \mathsf{cont}\,f(x_1);\ \mathsf{inr}\,z \mapsto q$$

where $q = (\mathsf{case}\ z\ \mathsf{of}\ \mathsf{inj}_1^{n-1}\,x_2 \mapsto p_2;\ \dots\ ;\mathsf{inj}_{n-1}^{n-1}\,x_n \mapsto p_n)$. According to (i), f is uniquely definable and thus we are done.

(iii) Let for some index k, $p_i \equiv \mathsf{cont}\,f(x)$ for $i \leq k$ and p_i does not contain f for $i > k$. If $k = 1$ and $n > 1$ then we arrive precisely at the situation captured by the previous clause and hence we are done. If $k = n = 1$ then ($**$) takes the form:

$$\mathsf{out}(f(x)) = \mathsf{do}\ x \leftarrow p; \mathsf{cont}\,f(x),$$

which can be transformed to:

$$\mathsf{out}(f(x)) = \mathsf{next}\,(\mathsf{do}\ x \leftarrow p; \mathsf{rest}\,x)\ \mathsf{is}\ \mathsf{rest}\,x \mapsto \mathsf{cont}\,f(x);\ \mathsf{done}\,x \mapsto \mathsf{stop}\,x$$

and hence we are done by **(corec)**. Suppose that $k > 1$, $n > 1$ and proceed by induction over k. Let

$$q = \mathsf{case}\ z\ \mathsf{of}\ \mathsf{inj}_1^{n-2}\,x_3 \mapsto p_3;\ \dots\ ;\mathsf{inj}_{n-2}^{n-2}\,x_{n-2} \mapsto p_{n-2}.$$

Then we have:

$$\begin{aligned}\mathsf{out}(f(x)) = {}& \mathsf{do}\ z \leftarrow p;\ \mathsf{case}\ z\ \mathsf{of}\ \mathsf{inl}\,x_1\ \mapsto \mathsf{cont}\,f(x_1);\\ &\qquad\qquad\qquad\qquad \mathsf{inr}\,\mathsf{inl}\,x_2\ \mapsto \mathsf{cont}\,f(x_2);\\ &\qquad\qquad\qquad\qquad \mathsf{inr}\,\mathsf{inr}\,z\ \mapsto q\\ = {}& \mathsf{next}\ \big(\mathsf{do}\ z \leftarrow p; \mathsf{case}\ z\ \mathsf{of}\ \mathsf{inl}\,x_1\ \mapsto \mathsf{cont}\,x_1;\\ &\qquad\qquad\qquad\qquad \mathsf{inr}\,\mathsf{inl}\,x_2\ \mapsto \mathsf{cont}\,x_2;\\ &\qquad\qquad\qquad\qquad \mathsf{inr}\,\mathsf{inr}\,z\ \mapsto \mathsf{stop}\,z\big)\ \mathsf{is}\\ &\mathsf{rest}\,x \mapsto \mathsf{cont}\,f(x);\ \mathsf{done}\,x \mapsto q\end{aligned}$$

and thus we are done by induction hypothesis.

(iv) Finally, we reduce the general claim to the case captured by the previous clause as follows. First observe that if neither of the p_i contains f then the solution is given by the equation:

$$f(x) = \mathsf{tuo}(\mathsf{do}\ z \leftarrow p; \mathsf{case}\ z\ \mathsf{of}\ \mathsf{inj}_1^n\, x_1 \mapsto p_1; \ldots; \mathsf{inj}_n^n\, x_n \mapsto p_n).$$

In the remaining case there should exist an index k and a permutation σ of numbers $1, \ldots, n$ such that for every $i \leq k$, $p_{\sigma(i)} \equiv \mathsf{rest}\, f(x)$ and for every $i > k$, $p_{\sigma(i)}$ does not contain f. By a slight abuse of notation we also use σ as function $A_1 + \ldots + A_n \to A_{\sigma(1)} + \ldots + A_{\sigma(n)}$ rearranging the components of coproducts in the obvious fashion. Then

$$\begin{aligned}\mathsf{out}(f(x)) = {} & \mathsf{do}\ z \leftarrow p; \mathsf{case}\ z\ \mathsf{of}\ \mathsf{inj}_1^n\, x_1 \mapsto p_1; \ldots; \mathsf{inj}_n^n\, x_n \mapsto p_n \\ = {} & \mathsf{do}\ z \leftarrow (\mathsf{do}\ z \leftarrow p; \mathsf{ret}\, \sigma(z)); \\ & \quad \mathsf{case}\ z\ \mathsf{of}\ \mathsf{inj}_{\sigma(1)}^n\, x_{\sigma(1)} \mapsto p_{\sigma(1)}; \ldots; \mathsf{inj}_{\sigma(n)}^n\, x_{\sigma(n)} \mapsto p_{\sigma(n)}\end{aligned}$$

and thus we are done by (iii). □

Proof of Theorem 5. We will need the following slight generalisation of Lemma 9.

Lemma 10. *Let f be a fresh functional symbol, i.e. $f \notin \Sigma$. Given appropriately typed programs $p, p_1, \ldots, p_n, q_1, \ldots, q_n$ such that for every i, p_i either does not contain f or is of the form $\mathsf{cont}\, f(q_i)$, there is a unique function f satisfying (∗∗).*

Proof. W.l.o.g. $q_i \equiv x$ whenever p_i does not contain f. We can rewrite (∗∗) to

$$\mathsf{out}(f(x)) = \mathsf{do}\ z \leftarrow q; \mathsf{case}\ z\ \mathsf{of}\ \mathsf{inl}_1^n\, x_1 \mapsto r_1; \ldots; \mathsf{inl}_1^n\, x_n \mapsto r_n$$

where $q = (\mathsf{do}\ z \leftarrow p; \mathsf{case}\ z\ \mathsf{of}\ \mathsf{inj}_1^n\, x_1 \mapsto \mathsf{ret}\, \mathsf{inj}_1^n\, q_1; \ldots; \mathsf{inj}_1^n\, x_n \mapsto \mathsf{ret}\, \mathsf{inj}_1^n\, q_n)$, $r_i = \mathsf{cont}(f(x))$ if $p_i \equiv \mathsf{cont}(f(q_i))$ and $r_i = p_i$ otherwise. Now we are done by Lemma 9. □

W.l.o.g. $n_1 = \ldots = n_k$: otherwise we replace every p_i by

$$\mathsf{do}\ z \leftarrow p_i; \mathsf{case}\ z\ \mathsf{of}\ \mathsf{inj}_1^{n_i}\, x_1 \mapsto \mathsf{inj}_1^n\, x_1; \ldots; \mathsf{inj}_{n_i}^{n_i}\, x_{n_i} \mapsto \mathsf{inj}_{n_i}^n\, x_n$$

where $n = \max_i n_i$ and complete the case in (5) arbitrary to match the typing. Observe that if the result types of f_i do not coincide, (5) falls in two mutually independent parts: once the result types of f_i, f_k are distinct, f_k can not appear in the definition of f_i and vice versa. Therefore, in the remainder we assume w.l.o.g. that f_i has type $A_i \to A$. Consider the following corecursive definition:

$$\begin{aligned}\mathsf{out}(F(y)) = \mathsf{do}\ v \leftarrow q; \mathsf{case}\ v\ \mathsf{of} & \\ & \mathsf{inj}_1^k\, z \mapsto (\mathsf{case}\ z\ \mathsf{of}\ \mathsf{inj}_1^n\, x_1 \mapsto q_1^1; \ldots; \mathsf{inj}_n^n\, x_n \mapsto q_n^1); \\ & \qquad \vdots \\ & \mathsf{inj}_k^k\, z \mapsto (\mathsf{case}\ z\ \mathsf{of}\ \mathsf{inj}_1^n\, x_1 \mapsto q_1^k; \ldots; \mathsf{inj}_n^n\, x_n \mapsto q_n^k)\end{aligned}$$

where

$$q = \mathsf{case}\ y\ \mathsf{of}\ \mathsf{inj}_1^k\, x_1 \mapsto (\mathsf{do}\ z \leftarrow p_i; \mathsf{ret}\,\mathsf{inj}_1^k\, z);\ \dots\ ; \mathsf{inj}_k^k\, x_k \mapsto (\mathsf{do}\ z \leftarrow p_k; \mathsf{ret}\,\mathsf{inj}_k^k\, z)$$

and the q_j^i are defined as follows: $q_j^i = \mathsf{cont}\, F(\mathsf{inj}_m^k\, x_j)$ if $p_j^i \equiv \mathsf{cont}\, f_m(x_j)$ and $q_j^i = p_j^i$ otherwise. By Lemma 10, it uniquely defines F. It is easy to calculate that by taking $f_i(x) = F(\mathsf{inj}_i^k(x))$ we obtain a solution of (5). Let us show it is also unique. Suppose that g_i is another solution. Then G defined by the equation

$$G(y) = \mathsf{case}\ y\ \mathsf{of}\ \mathsf{inj}_1^k\, x \mapsto g_1(x);\ \dots\ ; \mathsf{inj}_k^k\, x \mapsto g_k(x)$$

is easily seen to satisfy the same corecursive scheme as F and thus $F = G$. Therefore, for every i, $g_i(x) = G(\mathsf{inj}_i^k\, x) = F(\mathsf{inj}_i^k\, x) = f_i(x)$ and we are done. □

Proof of Theorem 4. *Soundness.* We only establish soundness of the rule **(corec)** since the remainder is more or less standard. Let $g = [\![\Gamma, x : A \rhd q : T(A+B)]\!]$, $h = [\![\Gamma, x : A \rhd p : T_\nu B]\!]$ and $f = T(\mathit{dist}) \circ \tau\langle \pi_1, g\rangle$. Then the top of **(corec)** can be rewritten to

$$h = R\pi_2 \circ F_f. \tag{2}$$

Let us show that the bottom of **(corec)** can be rewritten to

$$\alpha_{[\![B]\!]} \circ h = T(h + \pi_2) \circ f. \tag{3}$$

Indeed, we have:

$$\begin{aligned}
\alpha_{[\![B]\!]} \circ h &= [\![\mathsf{out}(p)]\!] \\
&= [\![\mathsf{do}\ z \leftarrow q;\ \mathsf{case}\ z\ \mathsf{of}\ \mathsf{inl}\, x \mapsto \mathsf{ret}\,\mathsf{inl}\, p;\ \mathsf{inr}\, y \mapsto \mathsf{ret}\,\mathsf{inr}\, y]\!] \\
&= T\big((h \circ (\pi_1\pi_1 \times \mathrm{id}) + \pi_2) \circ \mathit{dist} \circ \langle \mathrm{id}, \pi_2\rangle\big) \circ \tau\langle \mathrm{id}, g\rangle \\
&= T\big((h + \pi_2) \circ (\pi_1\pi_1 \times \mathrm{id} + \pi_1\pi_1 \times \mathrm{id}) \circ \mathit{dist} \circ \langle \mathrm{id}, \pi_2\rangle\big) \circ \tau\langle \mathrm{id}, g\rangle \\
&= T\big((h + \pi_2) \circ \mathit{dist} \circ (\pi_1\pi_1 \times \mathrm{id}) \circ \langle \mathrm{id}, \pi_2\rangle\big) \circ \tau\langle \mathrm{id}, g\rangle \\
&= T\big((h + \pi_2) \circ \mathit{dist} \circ (\pi_1 \times \mathrm{id})\big) \circ \tau\langle \mathrm{id}, g\rangle \\
&= T(h + \pi_2) \circ T(\mathit{dist}) \circ \tau\langle \pi_1, g\rangle \\
&= T(h + \pi_2) \circ f
\end{aligned}$$

In order to complete the proof, we are left to establish equivalence of (2) and (3). The proof of the implication (2) ⇒ (3) is as follows:

$$\begin{aligned}
\alpha_{[\![B]\!]} \circ h &= \alpha_{[\![B]\!]} \circ R\pi_2 \circ F_f \\
&= T(R\pi_2 + \pi_2) \circ \alpha_{[\![\Gamma]\!] \times [\![B]\!]} \circ F_f \\
&= T(R\pi_2 + \pi_2) \circ T(F_f + \mathrm{id}) \circ T(\mathit{dist}) \circ \tau\langle \pi_1, g\rangle \\
&= T(R\pi_2 \circ F_f + \pi_2) \circ T(\mathit{dist}) \circ \tau\langle \pi_1, g\rangle \\
&= T(h + \pi_2) \circ T(\mathit{dist}) \circ \tau\langle \pi_1, g\rangle \\
&= T(h + \pi_2) \circ f.
\end{aligned}$$

In order to prove (3), $\Rightarrow$ (2) let us assume (3). Observe that we can equivalently present the latter as $\alpha_{[\![B]\!]} \circ h = T(h + \mathrm{id}) \circ w$ where $w = T(\mathrm{id} + \pi_2) \circ f$. I.e. h satisfies the equation, which characterises F_f and thus $h = F_w$. We are left to show that $R\pi_2 \circ F_f$ also satisfies this equation. Indeed:

$$
\begin{aligned}
\alpha_{[\![B]\!]} \circ & R\pi_2 \circ F_f \\
&= T(R\pi_2 + \pi_2) \circ \alpha_{[\![\Gamma]\!] \times [\![B]\!]} \circ F_f \\
&= T(R\pi_2 + \pi_2) \circ T(F_f + \mathrm{id}) \circ T(\mathit{dist}) \circ \tau\langle \pi_1, g\rangle \\
&= T(R\pi_2 \circ F_f + \pi_2) \circ T(\mathit{dist}) \circ \tau\langle \pi_1, g\rangle \\
&= T(R\pi_2 \circ F_f + \pi_2) \circ f \\
&= T(R\pi_2 \circ F_f + \mathrm{id}) \circ T(\mathrm{id} + \pi_2) \circ f \\
&= T(R\pi_2 \circ F_f + \mathrm{id}) \circ w.
\end{aligned}
$$

We have thus $h = F_w = R\pi_2 \circ F_f$ and the proof is completed.

Completeness. By term model construction. □

Proof of Lemma 7. Suppose that $\{\phi\}p\{\psi\}$. Then we have:

$$
\begin{aligned}
&\mathsf{do}\ x \leftarrow \phi; y \leftarrow p; z \leftarrow \psi; \mathsf{ret}\langle x, y, z, x \Rightarrow z\rangle \\
&\quad = \mathsf{do}\ x \leftarrow \phi; y \leftarrow \mathsf{filter}(\phi, p, \psi); z \leftarrow \psi; \mathsf{ret}\langle x, y, z, x \Rightarrow z\rangle \\
&\quad = \mathsf{do}\ x \leftarrow \phi; y \leftarrow (\mathsf{do}\ x' \leftarrow \phi; y' \leftarrow p; z' \leftarrow \psi; \\
&\quad\quad \mathsf{if}(x' \Rightarrow z')\ \mathsf{then}\ \mathsf{ret}\, y'\ \mathsf{else}\ \varnothing); z \leftarrow \psi; \mathsf{ret}\langle x, y, z, x \Rightarrow z\rangle \\
&\quad = \mathsf{do}\ x \leftarrow \phi; x' \leftarrow \phi; y' \leftarrow p; z' \leftarrow \psi; \\
&\quad\quad \mathsf{if}(x' \Rightarrow z')\ \mathsf{then}\ \mathsf{do}\ y \leftarrow \mathsf{ret}\, y'; z \leftarrow \psi; \mathsf{ret}\langle x, y, z, x \Rightarrow z\rangle \\
&\quad\quad\quad\quad\quad\quad \mathsf{else}\ \mathsf{do}\ y \leftarrow \varnothing; z \leftarrow \psi; \mathsf{ret}\langle x, y, z, x \Rightarrow z\rangle \\
&\quad = \mathsf{do}\ x \leftarrow \phi; y \leftarrow p; z \leftarrow \psi; \mathsf{if}(x \Rightarrow z)\ \mathsf{then}\ \mathsf{ret}\langle x, y, z, x \Rightarrow z\rangle\ \mathsf{else}\ \varnothing \\
&\quad = \mathsf{do}\ x \leftarrow \phi; y \leftarrow p; z \leftarrow \psi; \mathsf{if}(x \Rightarrow z)\ \mathsf{then}\ \mathsf{ret}\langle x, y, z, \top\rangle\ \mathsf{else}\ \varnothing \\
&\quad = \mathsf{do}\ x \leftarrow \phi; y \leftarrow p; z \leftarrow \psi; \mathsf{ret}\langle x, y, z, \top\rangle
\end{aligned}
$$

On the other hand, provided the equation

$$
\begin{aligned}
&\mathsf{do}\ x \leftarrow \phi; y \leftarrow p; z \leftarrow \psi; \mathsf{ret}\langle x, y, z, x \Rightarrow z\rangle = \\
&\mathsf{do}\ x \leftarrow \phi; y \leftarrow p; z \leftarrow \psi; \mathsf{ret}\langle x, y, z, \top\rangle,
\end{aligned}
\tag{4}
$$

we have:

$$
\begin{aligned}
&\mathsf{filter}(\phi, p, \psi) \\
&\quad = \mathsf{do}\ x \leftarrow \phi; y \leftarrow p; z \leftarrow \psi; \mathsf{if}(x \Rightarrow z)\ \mathsf{then}\ \mathsf{ret}\, y\ \mathsf{else}\ \varnothing \\
&\quad = \mathsf{do}\, \langle x, y, z, v\rangle \leftarrow (\mathsf{do}\ x \leftarrow \phi; y \leftarrow p; z \leftarrow \psi; \mathsf{ret}(x, y, z, x \Rightarrow z)); \\
&\quad\quad \mathsf{if}\ v\ \mathsf{then}\ \mathsf{ret}\, y\ \mathsf{else}\ \varnothing \\
&\quad = \mathsf{do}\, \langle x, y, z, v\rangle \leftarrow (\mathsf{do}\ x \leftarrow \phi; y \leftarrow p; z \leftarrow \psi; \mathsf{ret}(x, y, z, \top)); \\
&\quad\quad \mathsf{if}\ v\ \mathsf{then}\ \mathsf{ret}\, y\ \mathsf{else}\ \varnothing \\
&\quad = \mathsf{do}\ x \leftarrow \phi; y \leftarrow p; z \leftarrow \psi; \mathsf{if}\ \top\ \mathsf{then}\ \mathsf{ret}\, y\ \mathsf{else}\ \varnothing
\end{aligned}
$$

$$= p.$$

By definition, this means validity of the Hoare triple $\{\phi\}p\{\psi\}$. $\square$